CHAPTER XX

Learning to Set Up Numerical Optimizations of Engineering Designs

Mark Schwabacher, Thomas Ellman, and Haym Hirsh
{schwabac, ellman, hirsh}@cs.rutgers.edu
Computer Science Department
Rutgers, The State University of New Jersey
New Brunswick, NJ 08903

ABSTRACT

Gradient-based numerical optimization of complex engineering designs offers the promise of rapidly producing better designs. However, such methods generally assume that the objective function and constraint functions are continuous, smooth, and defined everywhere. Unfortunately, realistic simulators tend to violate these assumptions, making optimization unreliable. Several decisions that need to be made in setting up an optimization, such as the choice of a starting prototype, and the choice of a formulation of the search space, can make a difference in how reliable the optimization is. Machine learning can help by making these choices based on the results of previous optimizations. We demonstrate this idea by using machine learning for four parts of the optimization setup problem: selecting a starting prototype from a database of prototypes, synthesizing a new starting prototype, predicting which design goals are achievable, and selecting a formulation of the search space. We use standard tree-induction algorithms (C4.5 and CART). We present results in two realistic engineering domains: racing yachts, and supersonic aircraft. Our experimental results show that using inductive learning to make setup decisions improves both the speed and the reliability of design optimization.

INTRODUCTION

Automated search of a space of candidate designs seems an attractive way to improve the traditional engineering design process. Each step of such automated search requires evaluating the quality of candidate designs, and for complex artifacts such as aircraft, this evaluation must be done by computational simulation.

Gradient-based optimization methods, such as sequential quadratic programming, are reasonably fast and reliable when applied to search spaces that satisfy their assumptions. They generally assume that the objective function and constraint functions are continuous, smooth, and defined everywhere. Unfortunately, realistic simulators tend to violate these assumptions. We call these assumption violations *pathologies*. Non-gradient-based optimization methods, such as simulated annealing and genetic algorithms, are better able to deal with search spaces that have pathologies, but they tend to require many more runs of the simulator than do the gradient based methods. We therefore would like to find a way to reliably use gradient-based methods in the presence of pathologies.

The performance of gradient-based methods depends to a large extent on choices that are made when the optimizations are set up, especially in cases where the search space has pathologies. For example, if a starting prototype is chosen in a less pathological region of the search space, the chance of reaching the optimum is increased. Machine learning can help by learning rules from the results of previous optimizations that map the design goal into these optimization setup choices. We demonstrate this idea by using machine learning for four parts of the optimization setup problem.

When designing a new artifact, it would be desirable to make use of information gleaned from past design sessions. Ideally one would like to learn a function that solves the whole design problem. The training data would consist of design goals and designs that satisfy those goals, and the learning algorithm would learn a function that maps a design goal into a design. We believe this function is too hard to learn. We therefore focused on improving optimization performance by using machine learning to make some of the choices that are involved in setting up an optimization. In the course of our work, we found parts of the optimization setup problem for which machine learning can help: selecting starting prototypes, predicting whether goals are achievable, and selecting formulations of the search space.

Our first effort was in the domain of the design of racing yachts of the type used in the America's Cup race. In this domain, we had success using a technique that we call *prototype selection* which maps the design goal into a selection of a prototype from a database of existing prototypes. We used C4.5, the standard tree-induction algorithm, in this work.

Our second effort was in the domain of the design of supersonic transport aircraft. We tried prototype selection in this domain, and found that it did not perform well, so we decided to try a new idea which we call *prototype syn-*

thesis. Prototype synthesis synthesizes a new prototype by mapping the design goal into the design parameters that define a prototype. It requires continuous-class induction, which is not available in C4.5; hence we used CART. We then realized that we could use the training data that we had collected for prototype synthesis to further enhance optimization performance using a new idea that we call *achievable goal prediction.* Achievable goal prediction uses inductive learning to predict whether a given design goal is achievable, before attempting to synthesize a starting prototype for the goal. Since this decision is discrete, rather than continuous, we used C4.5.

We then had the idea of recognizing when designs are at constraint boundaries, learning to predict this accurately, and using these predictions to reformulate the search space. We call this idea *formulation selection*. This prediction is discrete, so we used C4.5 to make it. We tested this idea in both the yacht and aircraft domains, and found it to be successful in both domains.

This chapter includes sections describing these four techniques for using machine learning to set up optimizations: prototype selection, prototype synthesis, achievable goal prediction, and formulation selection. Each section includes experimental results demonstrating that using the machine learning techniques improves the speed of optimization and/or the quality of the resulting designs.

INDUCTIVE LEARNING

The problem addressed by an inductive-learning system is to take a collection of labeled "training" data and form rules that make accurate predictions on future data. Inductive learning is particularly suitable in the context of an automated design system because training data can be generated in an automated fashion. For example, one can choose a set of training goals and perform an optimization for all combinations of training goals and library prototypes. One can then construct a table that records which prototype was best for each training goal.[1] This table can be used by the inductive-learning algorithm to generate rules mapping the space of all possible goals into the set of prototypes in the library. If learning is successful this mapping extrapolates from the training data and can be used successfully in future design sessions to map a new goal into an appropriate initial prototype in the design library.

The specific inductive-learning systems used in this work are C4.5 (Quinlan, 1993) (release 3.0, with windowing turned off) for problems requiring discrete-class induction, and CART[2] (Breiman, 1984) for problems requiring continuous-class induction. Both of these systems represent the learned knowledge in the form of decision trees. The approach taken by these systems is to find a small decision tree that correctly classifies the training data, then remove lower portions of the tree that appear to fit noise in the data. The resulting tree is then used as a decision procedure for assigning labels to future, unlabeled data.

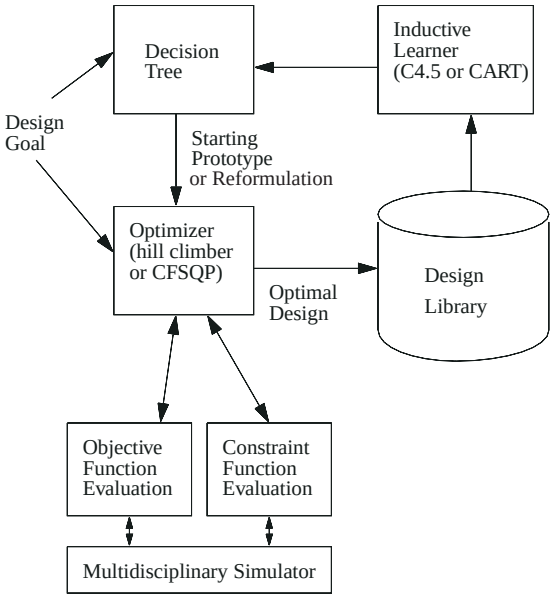

Figure 1. Design Associate block diagram

THE DESIGN ASSOCIATE

Our prototype-selection and formulation-selection techniques have been developed as part of the "Design Associate," a system for assisting human experts in the design of complex physical engineering structures (Ellman et al., 1992). Figure 1 shows a block diagram of the system's software architecture. The inductive learner learns from the design library a decision tree. Given a new design goal, the decision tree is used to map this design goal into a choice of starting prototype from the design library, or a choice of formulation of the search space. The optimizer optimizes this prototype for the new design goal, using the selected formulation. At each iteration of this optimization, the optimizer uses a multidisciplinary[3] simulator to evaluate the objective and constraint functions. At the end of the optimization, the new optimal design is added to the design library.

PROTOTYPE SELECTION

Many automated design systems begin by retrieving an initial prototype from a library of previous designs, using the given design goal as an index to guide the retrieval process (Sycara and Navinchandra, 1992). The retrieved prototype is then modified by a set of design modification operators to tailor the selected design to the given goals. In many cases the quality of competing designs can be assessed using domain-specific evaluation functions, and in such cases the design-modification process is often accomplished by an optimization method such as hill-climbing search (Ramachandran et al., 1992; Ellman et al., 1992).

Such a design system can be seen as a *case-based reasoning* system (Kolodner, 1993), in which the prototype-selection method is the *indexing* process, and the optimization method is the *adaptation* process.

In the context of such case-based design systems, the choice of an initial prototype can affect both the quality of the final design and the computational cost of obtaining that design, for three reasons. First, prototype selection may impact quality when the design process is guided by a nonlinear evaluation function with unknown global properties. Since there is no known method that is guaranteed to find the global optimum of an arbitrary nonlinear function (Schwabacher, 1996), most design systems rely on iterative local search methods whose results are sensitive to the initial starting point. Second, prototype selection may impact quality when the prototypes lie in disjoint search spaces. In particular, if the system's design modification operators cannot convert any prototype into any other prototype, the choice of initial prototype will restrict the set of possible designs that can be obtained by *any* search process. A poor choice of initial prototype may therefore lead to a suboptimal final design. Finally, the choice of prototype may have an impact on the time needed to carry out the design modification process — two different starting points may yield the same final design but take very different amounts of time to get there. In design problems where evaluating even just a single design can take tremendous amounts of time, we believe that selecting an appropriate initial prototype can be the determining factor in the success or failure of the design process.

To use inductive learning to form prototype-selection rules, we take as training data a collection of design goals, each labeled with which prototype in the library is best for that goal. "Best" can be defined to mean the prototype that best satisfies the design objectives, the prototype that results in the shortest design time, or the prototype that optimizes some combination of design quality and design time.

The Yacht Domain

We developed and tested our prototype selection methods in the domain of 12-meter racing yachts, which until recently was the class of yachts raced in America's Cup competitions.[4] An example of a 12-meter yacht is the *Stars & Stripes '87*, which is shown in Figure 2; a close-up of its hull and keel is shown in Figure 3.[5]

In the yacht domain, a design is represented by eight design parameters which specify the magnitude with which a set of geometric operators are applied to the B-spline surfaces (Rogers and Adams, 1990) of the starting prototype. The goal is to design the yacht which has the smallest course time for a particular wind speed and race course. Course time is evaluated using a "Velocity-Prediction Program" called "AHVPP" from AeroHydro, Inc., which is a marketed product used in yacht design (Letcher, 1991).

A search space is specified by providing an initial prototype geometry and a set of operators for modifying that prototype. Our current set of shape-

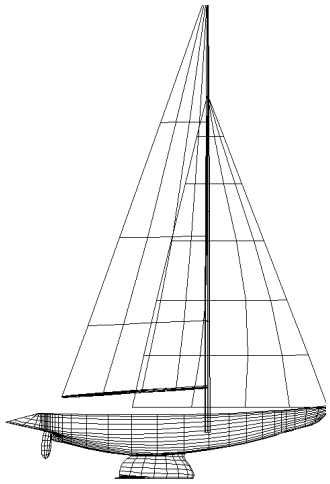

Figure 2. Stars & Stripes '87, winner of the 1987 America's Cup competition

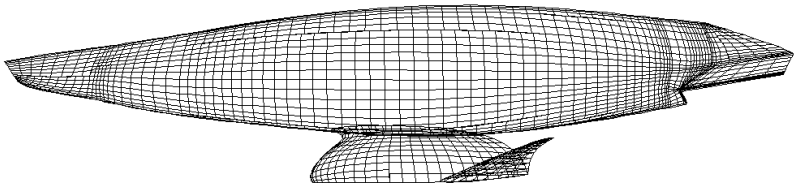

Figure 3. The hull and keel of Stars & Stripes '87.

modification operators was obtained by asking our yacht-design collaborators for an exhaustive list of all features of a yacht's shape that might be relevant to the racing performance of a yacht. These operators include

- Global-Scaling Operators: *Scale-X*, *Scale-Y* and *Scale-Z* change the overall dimensions of a racing yacht, by uniformly scaling all surfaces.
- Prismatic-Coefficient Operators: *Prism-X*, *Prism-Y* and *Prism-Z* make a yacht's canoe-body more or less streamlined, when viewed along the X, Y and Z axes respectively.
- Keel Operators: *Scale-Keel* and *Invert-Keel* change the depth and taper ratio of the keel respectively.

These eight operators represent a subset of the full set that were actually developed, focusing on a smaller set suitable for testing our prototype-selection methods.

Prototype Selection Results

We conducted several sets of experiments. In each case we compare our approach with each of four other methods:

1. **Closest goal**. This method requires a measure of the distance between two goals, and knowledge of the goal for which each prototype in the design library was originally optimized. It chooses the prototype whose original goal has minimum distance from the new goal. Intuitively, in our yacht-design problem this method chooses a yacht designed for a course and wind speed most similar to the new course and wind speed.
2. **Best initial evaluation**. This method requires running the evaluation function on each prototype in the database. It chooses the prototype that, according to the evaluation function, is best for the new goal (before any operators have been applied to the prototype). In the case of our yacht-design problem this corresponds to starting the design process from whichever yacht in the library is fastest for the new course and wind speed.
3. **Most frequent class**. This is actually a very simple inductive method that always chooses a fixed prototype, namely the one that is most frequently the best prototype for the training data.
4. **Random**. This method involves simply selecting a random element from the design library, using a uniform distribution over the designs.

We compare these methods using two different evaluation criteria:

1. **Error rate**. How often is the wrong prototype selected?
2. **Course-time increase**. How much worse is the resulting average course time than it would be using the optimal choice that an omniscient selection would make?

In our experiments we focus primarily on the question of how well our inductive-learning prototype-selection method handles problems where the prototypes lie in disjoint search spaces. Our experiments therefore explore how prototype selection affects the quality of the final design.

For the prototype selection experiments in the yacht domain, we used the Rutgers Hill-climber as our optimizer. It is an implementation of steepest-descent hill-climbing, that has been augmented so as to allow it to "climb over" bumps in the surface defined by the objective function that have less than a certain width or a certain height.

For our first set of experiments we created a database of four designs that would serve as our sample prototype library (and thus also serve as the class labels for the training data given to our inductive learner). To simulate the effect of having each prototype define a different space, the design library was created by starting from a single prototype (the Stars and Stripes '87) and optimizing for four different goals using all eight of our design-modification operators. All subsequent design episodes used only four of the eight operators, so that each yacht would define a separate space.[6]

TABLE 1. *A portion of the input to C4.5 for prototype selection in the yacht domain.*

Long-Leg	Short-Leg	Wind speed	Initial-Design
180	0	8	Design 1
180	0	10	Design 2
180	0	12	Design 2
180	0	14	Design 2
180	0	16	Design 2
180	90	8	Design 1
180	90	10	Design 4
180	90	12	Design 4
180	90	14	Design 4
180	90	16	Design 1

```
long-leg <= 90 :
|   windspeed > 10 : Design-1
|   windspeed <= 10 :
|   |   short-leg <= 90 : Design-1
|   |   short-leg > 90 : Design-2
long-leg > 90 :
|   windspeed > 14 : Design-2
|   windspeed <= 14 :
|   |   windspeed <= 10 : Design-4
|   |   windspeed > 10 : Design-4
```

Figure 4. Example of a prototype-selection decision tree generated by C4.5.

We defined a space of goals to use in testing the learned prototype-selection rules. Each goal consists of a wind speed and a race course, where the wind speed is constrained to be 8, 10, 12, 14, or 16 knots, the race course is constrained to be 80% in one direction, and 20% in a second direction, and each direction is constrained to be an integer between 0 and 180 degrees. This space contains 162,900 goals.

To generate training data we defined a set of "training goals" that spans the goal space. This smaller set of goals was defined in the same fashion as for the testing set of goals except that the directions in the race course are restricted to be only 0, 90, or 180 degrees, yielding a smaller space of 30 goals. To label the training data we attempted to find designs for each of the 30 goals starting from each of the four prototypes using the restricted set of operators, and determined which starting point was best.

To generate test data we randomly selected ten "testing goals" from the goal space. We then generated designs starting from each of the four prototypes in the database for each of these testing goals to determine which prototype was best, as well as to determine how much of a loss in course time each incorrect

TABLE 2. *Comparison of prototype-selection methods when trained on a set of goals that spans the goal space, using AHVPP.*

Method	Error Rate	Course-Time Increase (sec)
Inductive Learning	30%	24
Most Frequent Class	70%	47
Random Guessing	75%	62
Best Init Eval	70%	64
Closest Goal	70%	78

selection would impose. Table 1 shows a portion of the input to C4.5, and Figure 4 gives an example of a decision tree output by C4.5. Table 2 compares the results using C4.5 with the other prototype-selection methods. (Since there are four prototypes, one would expect random guessing to get 75% of the test examples wrong.)

In this experiment, the inductive method (C4.5) performed better than the other methods on both measures of performance. Moreover, we were particularly surprised by how poorly the non-inductive prototype-selection methods (closest goal and smallest initial evaluation) performed — our expectation was that the prototypes chosen by these methods would be close in "design space" to the optimal final design, thus yielding better final designs than starting from the other prototypes.

After studying these results we generated two new hypotheses for why these two prototype-selection methods did not work well. The first is that the shape of the design space may be such that there is little relationship between the distance between two designs and the ability of the hill-climber to climb from one design to the other. If the space contains "bumps" or "ridges" over which the hill-climber cannot climb, then it might be more important for the initial prototype to be on the "right side" of a bump or a ridge than for it to be close to the optimal point. Our second new hypothesis was that some of the prototypes in the database may be "bad" prototypes. This could be the case if the hill-climber got stuck at a local (non-global) optimum during the run that produced the prototype. This latter hypothesis was supported by the fact that one of the four prototypes was never found to be a good starting point for any of the 30 goals in the training data (not even the goal for which it was supposedly optimal, since it wound up being a local optimum and starting from another prototype yielded a superior result). In a realistic design scenario, when there is no control over the source of a design library, there could easily be "bad" prototypes included. Unlike the non-inductive prototype-selection methods, the inductive methods learn to avoid the bad prototypes.

We performed some experiments to test our first new hypothesis that the closest-goal and smallest-initial-evaluation methods performed poorly because of the "bumps" in the evaluation function. We repeated the earlier experiments

TABLE 3. *Comparison of prototype-selection methods when trained on a set of training examples that spans the goal space, using the simplified VPP.*

Method	Error Rate	Course-Time Increase (sec)
Best Init Eval	12%	26
Inductive Learning	37%	57
Closest Goal	40%	76
Most Frequent Class	45%	175
Random Guessing	75%	257

using a simplified, "smooth" velocity prediction program, called "RUVPP," that we developed at Rutgers. RUVPP differs from the more complex AHVPP in several respects. To begin with, RUVPP represents a yacht as a list of major geometric dimensions such as length, depth, and beam, rather than B-spline surfaces. Furthermore, RUVPP embodies a number of simplifying assumptions about the physics of sailing that are not made in AHVPP. Nevertheless, the simple version, RUVPP, is useful for two reasons: RUVPP is much faster to execute than AHVPP, and RUVPP has fewer of the bumps and ridges that appear in AHVPP. We therefore expect that a hill-climbing search algorithm is less likely to get stuck on the wrong side of a bump or ridge when the simple version, RUVPP, is used as an evaluation function. Table 3 presents the results of experiments comparing the performance of inductively learned prototype-selection rules to the other prototype-selection methods, repeating our earlier experiments, but using RUVPP as the evaluation function, and using forty random test cases instead of just ten.

Because RUVPP is much faster than AHVPP, we conducted additional supporting experiments to test our first new hypothesis, to see if using a spanning set of goals as training data was significant for our results. In particular, rather than using inductive learning on a set of goals that span the space of possible goals, we also performed experiments where C4.5 was trained on a random sample of goals selected from the same space as the testing data. This was done using ten trials of four-fold cross-validation on a set of forty random goals. Each such trial involves randomly dividing the data into four sets of size ten, using three of the sets for training data and the remaining one as testing. This is repeated four times, using each ten-element set once for testing, and this process was repeated ten times with different random partitionings of the data. Table 4 reports the results of these experiments.

Consistent with our first new hypothesis, the closest-goal and best-initial-evaluation methods both did much better in both cases with the simplified VPP than they did with AHVPP, while C4.5 did about the same as it had done before. We believe that because the simplified VPP is much smoother than AHVPP, the hill-climber is much less likely to get stuck, so that the distance in goal space or

TABLE 4. *Comparison of prototype-selection methods when trained and tested on random goals, using cross-validation and the simplified VPP.*

Method	Error Rate	Course-Time Increase (sec)
Best Init Eval	12%	26
Inductive Learning	30%	35
Closest Goal	40%	76
Most Frequent Class	45%	175
Random Guessing	75%	257

TABLE 5. *Comparison of prototype-selection methods when trained on a set of goals that span the space, using the simplified VPP, and a "bad" prototype in the database.*

Method	Error Rate	Course-Time Increase (sec)
Best Init Eval	10%	80
Inductive Learning	30%	82
Closest Goal	32%	89
Most Frequent Class	45%	171
Random Guessing	75%	348

the difference in initial evaluation becomes much more relevant when choosing a prototype. In fact, the improvement in the best-initial-evaluation method was so great that it significantly outperformed the inductive method.

We performed another set of experiments to test our second new hypothesis of why the closest-goal and smallest-initial-evaluation method performed so poorly using AHVPP, namely that they were unable to avoid the "bad" proto-

TABLE 6. *Comparison of prototype-selection methods when trained and tested on a set of random goals, using cross-validation, the simplified VPP, and a "bad" prototype in the database.*

Method	Error Rate	Course-Time Increase (sec)
Inductive Learning	19%	38
Best Init Eval	10%	80
Closest Goal	32%	89
Most Frequent Class	45%	171
Random Guessing	75%	348

type in the database. We repeated our preceding experiments using the simplified VPP, except that we intentionally put a "bad" prototype into the database. To generate a bad prototype, we started with the Stars and Stripes '87, and added a random number between -0.2 and +0.2 to each of the operator parameters. We then randomly chose one of the four prototypes in the database to replace with the bad prototype (but we left the class label the same). The results of repeating the experiments with the bad prototype in the database are presented in Table 5 for training on goals that span the space, and Table 6 for training on random goals.

Consistent with our second new hypothesis, C4.5's ability to avoid the "bad" prototype improved its performance relative to the other methods. When trained on the spanning goals, C4.5 performed only slightly worse than the smallest-initial-evaluation method. When trained on the random goals, C4.5 performed markedly better than any other method as measured by average course-time increase, although the smallest-initial-evaluation method had a lower error rate. This apparent anomaly can be explained as follows: The "bad" prototype was very bad, so that choosing it even a few times resulted in large increases in average course time. C4.5 never chose the bad prototype. The best-initial-evaluation method occasionally chose the bad prototype, so that even though it chose the best prototype more frequently than C4.5, the few times when it chose the bad prototype worsened its average course-time increase.

The Cost of Learning

One important question to answer is whether the inductive prototype-selection method is worth the considerable "off-line" expense of collecting training data — every training example requires one design run for each design in the prototype library. An alternative, possibly cheaper method would be to take an "on-line" approach: for each new design problem optimize starting from every prototype in the database, and then use whichever of the resulting designs is the best.

If the quality of the final design is extremely important and there is ample CPU time available, this "exhaustive" method is the one to use (out of the methods listed in Table 2). On the other hand, if limiting CPU time is important, our inductive learning method becomes cost effective when the computational expense of learning can be amortized over a sufficiently large number of new design goals. More specifically, the inductive prototype-selection method is less expensive than the exhaustive method whenever the number of hill-climbing runs taken by the inductive approach is less than the number of runs taken by the exhaustive approach, i.e., $TP + G < PG$ or

$$G > \frac{T}{1 - \frac{1}{P}}$$

where T is the number of training examples, P is the number of prototypes in the database, and G is the number of new goals for which prototypes need to be

Phase	Mach	Altitude m	Altitude ft	Duration (min)	comment
1	0.227	0	0	5	"takeoff"
2	0.85	12 192	40 000	50	subsonic cruise (over land)
3	2.0	18 288	60 000	225	supersonic cruise (over ocean)

capacity: 70 passengers.

TABLE 7. *Mission specification for aircraft in Figure 5*

selected. (When using the inductive prototype-selection method, TP is the cost of generating the training data, and G is the cost of performing optimizations for the new goals. When using the exhaustive method, each prototype in the database must be optimized for each new goal, at a cost of PG.) In all of the experiments that we performed, there were four prototypes and 30 training examples, so our inductive approach will be less expensive than the exhaustive approach as long as at least 40 out of the more than 150,000 remaining design goals must be attempted.

When doing prototype synthesis rather than prototype selection, it is not necessary to collect training data in which each prototype in a database is used as a starting point of an optimization for each of a collection of goals. (Prototype synthesis takes as training data the optimal design parameters for each goal, rather than the selection of the best prototype from a database for each goal.) Instead, any optimizations that have been previously done (within the same goal space) can be used as training data. Hopefully, such data will already exist in a design library, so additional optimizations will not be needed to generate training data. Prototype synthesis is further described in the next section.

PROTOTYPE SYNTHESIS AND ACHIEVABLE GOAL PREDICTION

Prototype synthesis uses continuous-class induction (also known as regression) to map the design goal directly into the design parameters that define a new prototype, instead of selecting an existing prototype from a database. What is learned is not a set of rules for selecting a prototype, but rather a set of functions that map the design goal into the design parameters. We performed some experiments to test prototype synthesis in the domain of supersonic transport aircraft design.

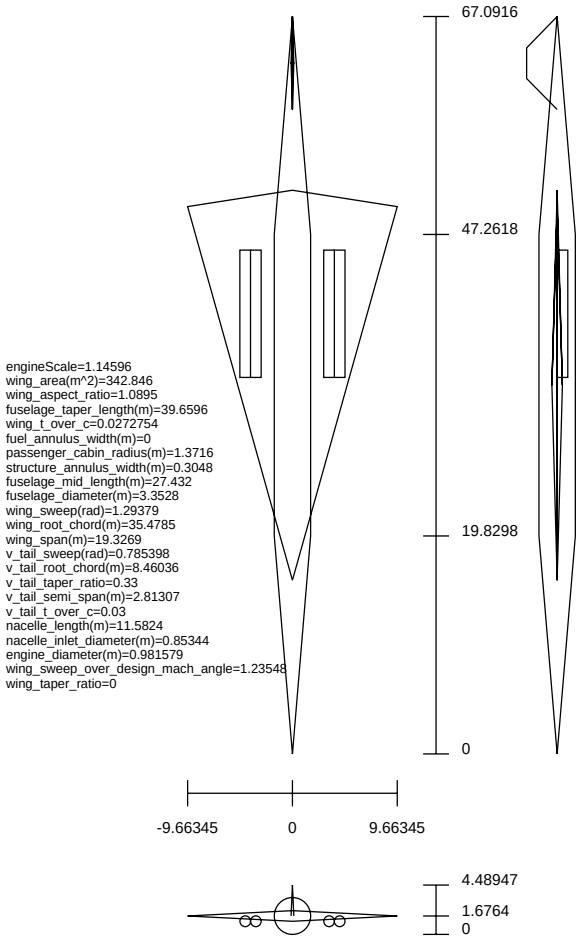

Figure 5. Supersonic transport aircraft designed by our system (dimensions in meters)

The Aircraft Domain

Figure 5 shows a diagram of a typical airplane automatically designed by our software system to fly the mission shown in Table 7. The optimizer attempts to find a good aircraft conceptual design for a particular mission by varying major aircraft parameters such as wing area, aspect ratio, engine size, etc; using a numerical optimization algorithm. The optimizer evaluates candidate designs using a multidisciplinary simulator. In our current implementation, the optimizer's goal is to minimize the takeoff mass of the aircraft, a measure of merit commonly used in the aircraft industry at the conceptual design stage. Takeoff mass is the sum of fuel mass, which provides a rough approximation of the operating cost of the aircraft, and "dry" mass, which provides a rough approximation of the cost of building the aircraft. The simulator computes the takeoff mass of a particular aircraft design for a particular mission as follows:

1. Compute "dry" mass using historical data to estimate the weight of the

aircraft as a function of the design parameters and passenger capacity required for the mission.

2. Compute the landing mass $m(t_{\text{final}})$ which is the sum of the fuel reserve plus the "dry" mass.
3. Compute the takeoff mass by numerically solving the ordinary differential equation

$$\frac{dm}{dt} = f(m, t)$$

which indicates that the rate at which the mass of the aircraft changes is equal to the rate of fuel consumption, which in turn is a function of the current mass of the aircraft and the current time in the mission. At each time step, the simulator's aerodynamic model is used to compute the current drag, and the simulator's propulsion model is used to compute the fuel consumption required to generate the thrust which will compensate for the current drag.

A complete mission simulation requires about 1/4 second of CPU time on a DEC Alpha 250 4/266 desktop workstation.

The numerical optimizer used in the prototype synthesis experiments is CFSQP (Lawrence et al., 1995)[7], a state-of-the-art implementation of the Sequential Quadratic Programming (SQP) method. SQP is a quasi-Newton method that solves a nonlinear constrained optimization problem by fitting a sequence of quadratic programming problems[8] to it, and then solving each of these problems using a quadratic programming method. We have supplemented CFSQP with *rule-based gradients* (Schwabacher and Gelsey, 1997) and *model constraints* (Gelsey et al., 1998).

Because the search space has many local optima, we use a technique that we call "random multistart" to attempt to find the global optimum. In an n-point random multistart, the system randomly generates starting points within a particular box until it finds n evaluable points[9], and then performs an SQP optimization from each of these points. The best design found in these n optimizations is taken to be the global optimum.

In the airframe domain, the design goal is to minimize take-off mass (a rough estimate of life-cycle cost) for a specified mission. We defined the following space of missions:

```
distance between 1609 km (1000 miles)
  and 16 090 km (10 000 miles)
percentage over land between 0 and 100%
mach number over land of 0.85
altitude over land 12 192 m (40 000 ft)
mach number over water between 1.5 and 2.2
altitude over water 18 288 m (60 000 ft)
optional takeoff phase, no climb phase
```

A mission within this space can be represented using three real numbers (distance, percentage over land, and mach number) and one Boolean value

```
distance > 14 456 km (8982.46 miles): infeasible
distance <= 14 456 km (8982.46 miles):
|   distance <= 10 276 km (6384.94 miles): feasible
|   distance > 10 276 km (6384.94 miles):
|   |   overland <= 23.6023% : feasible
|   |   overland > 23.6023% : infeasible
```

Figure 6. Learned decision tree for deciding if a mission is feasible.

(whether the takeoff phase is included). We generated 100 random missions as follows: The distance and mach number were uniformly distributed over their possible ranges. There was a 1/3 probability of having the mission entirely over land, a 1/3 probability of having it entirely over water, and a 1/3 probability of having the percentage over land uniformly distributed between 0 and 100%. There was a 1/2 probability of including the takeoff phase.

Achievable Goal Prediction

In order to generate training data to test our techniques in the airframe domain, we performed a 10-point random multistart CFSQP optimization for each of the 100 random missions. We found that for many of these missions, CFSQP was unable to find a feasible design in any of the ten runs — that is, it was unable to design a plane that could fly the mission. It occurred to us that it would be valuable if we could predict in advance whether a given mission was achievable, so that we could avoid attempting to synthesize prototypes for infeasible missions. We hypothesized that C4.5 would be able to make this prediction, and that it would be able to do so with greater accuracy than MFC.

To test this new *achievable goal prediction* idea, we trained C4.5 on a set of training examples showing whether each of our 100 missions was feasible. It produced the decision tree in Figure 6. This decision tree shows that missions are infeasible if they are very long, or if they are moderately long and have a significant portion over land. Further analysis revealed that building a plane to fly such a mission would require an engine larger than the largest engine that we allowed. Our upper bound on engine size can be considered to be representative of the largest commercially available engine.

Tenfold cross validation showed that C4.5 has a 4% error rate on this learning task, compared with 50% for random guessing and 24% for most frequent class. The decision tree of figure 6 can be used to predict, without doing any simulation or optimization, whether a new proposed mission is feasible.

Prototype Synthesis

In order to map the new mission into the numerical design parameters that define a prototype, we need to use *continuous-class induction* (which is also

TABLE 8. *Accuracy of CART in predicting each design parameter in the airframe domain.*

Design Parameter	Relative RMSE
engine size	0.65
wing area	0.59
wing aspect ratio	0.06
fuselage taper length	0.07
effective structural thickness over chord	0.08
wing sweep over design mach angle	0.08
wing taper ratio	0.21
fuel annulus width	1.02

TABLE 9. *Comparison of prototype-synthesis methods.*

Method	Success	Cost (number of simulations)
CART	13/16	7394
mean	14/16	11963
1 random	8/16	16893
2 random	13/16	33883
3 random	14/16	47395

known as regression). We used CART (Classification And Regression Trees), which builds a "regression tree" that has a numerical constant at each leaf (Breiman, 1984). We trained CART on the 100 randomly generated training goals as follows: For each design parameter, we gave CART a set of training data, where each item in the training data included the goal and the optimal value of the design parameter. CART thus generated a set of trees to map the design goal into a set of design parameters that we hope will be near the optimal values for that goal. Table 8 shows the root mean squared error in CART's prediction of each design parameter, relative to the error of "constant regression," which always uses the mean of the training data. A value less than one in this table indicates that CART's prediction was more accurate than that of constant regression. Our expectation that these relative errors would be low was confirmed for all of the parameters except fuel annulus width.

We performed a set of experiments to test whether using these trees to do prototype synthesis would produce better optimization performance than using the mean prototype or a random prototype. We used 25 randomly generated testing goals. Table 9 compares using the prototypes synthesized by CART with using a 1-, 2-, or 3-point random multistart, or always using the prototype which is the mean of all the optimized prototypes in the training data. Of the 25 randomly generated test goals, 16 were feasible. The "success" column shows the number of optimizations that came within 1% of the point that we believe to be the global optimum.[10] Some of the failures occurred because the

TABLE 10. *Performance of one random probe, averaged over ten trials.*

Measure	Success	Cost (number of simulations)
Mean	8.8/16	15062
Standard Deviation	1.9/16	3102

learning method produced an unevaluable prototype that could not be simulated, and therefore could not be optimized. Other failures occurred because the optimizer, when started from the synthesized point, failed to get within 1% of the apparent global optimum. The "cost" column shows the total number of simulations used in the 16 optimizations. Using the mean prototype instead of a single random prototype resulted in much greater success, at 33% lower cost. Using CART produced a success rate about the same as using the mean prototype, with an additional 38% cost reduction. Using a 2-point random multistart produced the same success rate as using CART, but it required more than four times as many simulations.

To test the significance of the result that CART performed better than one random probe, we repeated the one-random-probe test ten times, with ten different seeds to the random number generator. The mean and standard deviation of the success rate and cost are shown in Table 10. CART's success rate was more than two standard deviations higher than that of one random probe, and its cost was more than two standard deviations lower than that of one random probe.

FORMULATION SELECTION

Besides the selection of a starting prototype, another important decision in setting up an optimization is the decision on how to formulate the search space. This decision can substantially affect the performance of the optimizer in two ways. First, using a lower-dimensional formulation of the search space makes optimization faster, since each gradient computation requires fewer runs of the simulator, and the distance in design space from the starting point to the optimum is smaller. Second, different formulations of the search space can result in different degrees of "smoothness" of the search space, which can impact not only the speed of the optimizer, but also the ability of the optimizer to get to the optimum, and therefore the quality of the resulting designs.

We present a method of reformulation called "constraint incorporation," which reduces the dimensionality of the search space and increases its smoothness by incorporating constraints into the search space.

Traditionally, numerical optimization has dealt with explicit, "hard" constraints. The optimizer assumes that these constraints can never be violated. A

hard constraint can be expressed as

$$f(x_1, x_2, \ldots, x_n) \leq k$$

(Here $x_1, x_2, \ldots, x_n$ are the *design parameters* that represent the design.) The constraint is said to be *inactive* if $f(x_1, x_2, \ldots, x_n) < k$, *active* if $f(x_1, x_2, \ldots, x_n) = k$, and *violated* if $f(x_1, x_2, \ldots, x_n) > k$. Hard constraints can result from the laws of physics, for example.

Another type of constraint is the "soft" constraint, for which there is some sort of known penalty for violating the constraint. A soft constraint can be expressed as

if $f(x_1, x_2, \ldots, x_n) > k$ then apply penalty $P(x_1, x_2, \ldots, x_n)$

These usually arise from human-written laws, such as regulations specifying a monetary penalty for exceeding a certain noise level. In either case, if it is known that the constraint will be active at the optimal design point, and the constraint function f is invertible, then the constraint can be *incorporated* into the search space by using the inverse of f to eliminate one of the design parameters. This incorporation is done by making the inequality constraint into an equality constraint, and then solving for one of the design parameters in terms of the other design parameters. (Papalambros and Wilde, 1988) describe how monotonicity knowledge can be used to determine that certain constraints will be active at the optimum. Incorporating these constraints produces a new search space with lower dimensionality, since the incorporation eliminates a design parameter, and greater smoothness, since the incorporation eliminates the "ridge" (or non-smoothness) in the search space caused by the "if" statement in the constraint. If there are n constraints that can be incorporated in this way, then there are 2^n possible formulations that can be produced by incorporating different subsets of constraints.

Constraint activity depends on the goal (some constraints are active at the optimum for only some design goals), for two reasons: First, the constraint thresholds are part of the design goal. Second, different design goals will result in different optimal values of the design parameters on which the constraint functions depend.

Because constraint activity depends on the goal, different formulations of the search space are appropriate for different design goals. We describe a way in which inductive learning can be used to map the design goal into the appropriate formulation.

To use inductive learning to form formulation-selection rules, we take as training data a collection of design goals, each labeled with the set of constraints that are active (within a threshold) at the optimal design point. We run the inductive learner once for each constraint, producing for each constraint a set of rules that can be used to predict whether the constraint will be active for new design goals.

The training data can be generated in an automated fashion. For example, one can choose a set of training goals and perform an optimization for each

goal. One can then evaluate each constraint function for each optimal design, and then construct a table that records which constraints were active (within a threshold) for each training goal. This table can be used by the inductive-learning algorithm to generate a set of rules for each constraint, mapping the space of all possible goals into a prediction of whether or not that constraint will be active at the optimal design point for that goal. If learning is successful, these mappings extrapolate from the training data and can be used successfully in future design sessions to map a new goal into an appropriate formulation.

Formulation Selection Results in Yacht Domain

We performed some experiments to test the performance of formulation selection in the yacht domain. In the experiments described in this subsection, we used CFSQP as the optimizer, with *course-time*, computed by RUVPP, as the objective function, and with one explicit, nonlinear, "hard" model constraint. This constraint specifies that the mass of the yacht, before adding any ballast, must be less than or equal to the mass of the water that it displaces. (In other words, the boat must not sink.)

Yachts entered in the 1987 America's Cup race had to satisfy a hard constraint known as the 12-Meter Rule (IYRU, 1985). Instead of using this rule as an explicit constraint, we incorporated it into the search space. (How we incorporated it is described below.) The basic formula in the rule is:

$$\frac{length - freeboard + \sqrt{sailarea}}{2.37} \leq 12m$$

In addition to the basic formula, the rule contains several soft constraints, along with associated penalties for violating these constraints. These soft constraints are:

- draft constraint
- beam constraint
- displacement constraint
- winglet span constraint

For example, the *beam constraint* states

if $beam < 3.6\mathrm{m}$, then add four times the difference to $length$

While constructing the simulator, we used a reasoning process similar to that described in (Papalambros and Wilde, 1988) to determine that the constraint described by the basic formula of the 12-Meter Rule, above, will always be active, since the objective function being minimized, *course-time*, is monotonically non-increasing in *sail-area*,[11] and the left-hand-side of the constraint is monotonically increasing in *sail-area*. We therefore *incorporated* this constraint into the simulator by solving for *sail-area* in terms of the other design parameters. So, for example, when the optimizer makes $length$ bigger, *sail-area* is automatically made smaller. In addition, because we also implemented the soft

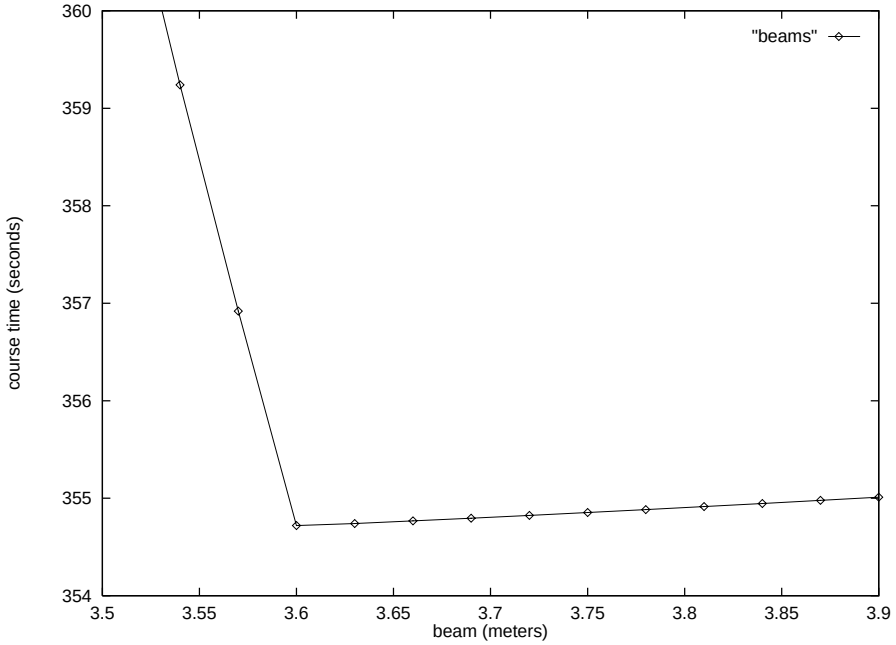

Figure 7. The nonsmoothness in the search space caused by the beam constraint.

constraints as penalty functions, reducing $beam$ beyond $3.6m$ causes the quantity *length* in the formula to increase, which causes *sail-area* to decrease.[12]

Because the beam constraint contains an *if* statement, this incorporation causes a nonsmoothness in *course-time* as a function of $beam$. That is, there is a discontinuity in the first derivative of *course-time* with respect to $beam$. Figure 7 illustrates this nonsmoothness by showing the cross-section of the search space corresponding to the $beam$ design parameter.[13] This nonsmoothness can cause a gradient-based optimizer such as CFSQP to get stuck, and to fail to get to the optimum.

For many design goals, the optimal design is right on the constraint boundary. The optimal beam is often 3.6 m. If we expect the optimal beam to be 3.6 m, then we can incorporate the beam constraint into the operators. In the case of the beam constraint, this incorporation is trivial — we simply set $beam$ to 3.6 m and leave it there. For other constraints, the incorporation is more complicated. For example, there is a constraint that specifies a penalty if $displacement$ does not vary with a certain cubic polynomial in $length$. $Displacement$ is not a design parameter; rather, it is a quantity computed from all of the design parameters. In order to incorporate the displacement constraint, we used Maple (Char et al., 1992), a symbolic algebra package, to invert the displacement formula, and created a new set of operators that vary certain parameters while maintaining $displacement$ at the minimum displacement allowed by the constraint. For still-more-complicated constraints, it might not be possible to invert the constraint function using Maple; it might therefore be necessary for the opera-

tors to contain numerical solvers that find the right values of the incorporated design parameters so as to put the design on the constraint boundary.[14]

We created operators to incorporate all four of the above-listed 12-Meter Rule constraints: the draft constraint, the beam constraint, the displacement constraint, and the winglet constraint. Using these operators, we are able to either incorporate or not incorporate each of these four constraints independently. We thus defined a set of sixteen (2^4) possible formulations of the search space. From our initial experiments with these operators, we determined empirically that incorporating the draft constraint substantially improved the reliability and speed of optimization for any design goal. We therefore decided to always incorporate the draft constraint, leaving us with a space of eight possible formulations that we used in the experiments described below.

Having defined eight formulations of the search space, we used inductive learning to decide, based on the design goal, which formulation to use. As training data, we used 100 previous optimizations. The optimizer failed for one of these goals, so we used the remaining 99 goals as training data in the results that follow. For each previous optimization, we evaluated each 12-Meter Rule constraint function at the optimum, and determined if the constraint was active (within a tolerance). Each of these previous optimizations had as its design goal minimizing course time for a single-leg race course, which can be represented using two numbers: the wind speed, and the heading (the angle between the yacht's direction and the wind direction). The design goal can therefore be represented using these two numbers. We ran the inductive learner once for each of the three constraints. Each time, the inductive learner was provided with a set of triples: the wind speed, the heading, and a ternary value indicating whether the constraint was inactive, active, or violated. One of the constraints was violated at the optimum in 10 of these optimizations. Figure 8 gives an example of a decision tree output by C4.5. This decision tree predicts whether the displacement constraint will be active at the optimum, based on the design goal. By running a new design goal down three decision trees, one for each of the three constraints that can be incorporated, the system can make predictions of whether each constraint will be active at the optimum. These three yes/no predictions directly map into one of the eight (2^3) formulations of the search space.

We used C4.5 to perform tenfold cross-validation, and obtained the error rates shown in Table 11. Here we compare the error rates of C4.5 with and without pruning, and of C4.5rules, a variant of C4.5 that extracts rules from the trees, with the expected error rate of random guessing (which is two-thirds since there are three classes from which to guess), and the error rate of the Most Frequent Class (MFC) learning method. MFC always chooses the class that occurs most frequently in the training data. In this case, that means that it always chooses the same formulation, namely the one that is most often the best formulation in the training data.

As Table 11 shows, C4.5 with pruning performed slightly better than C4.5 without pruning or C4.5rules (and so in our further experiments reported below

```
heading <= 109 :
|   windspeed <= 6.3 : active
|   windspeed > 6.3 :
|   |   windspeed > 8.2 : violated
|   |   windspeed <= 8.2 :
|   |   |   heading <= 65 : violated
|   |   |   heading > 65 : active
heading > 109 :
|   windspeed > 11.5 : active
|   windspeed <= 11.5 :
|   |   heading <= 135 : active
|   |   heading > 135 : inactive
```

Figure 8. Learned decision tree for the displacement constraint.

TABLE 11. *Cross-validated error rates for selecting whether to incorporate each constraint.*

method	Beam	Displacement	Winglet
C4.5 w/ pruning	11.1%	15.1%	7.0%
C4.5 w/o pruning	11.1%	15.1%	10.0%
C4.5rules	11.1%	15.1%	10.0%
MFC	33.3%	53.5%	13.1%
Random	66.7%	66.7%	66.7%

we use only C4.5 with pruning), and all three substantially outperformed MFC, which in turn substantially outperformed random guessing.

These results are for error rates, the proportion of cases where learning makes an incorrect guess. A more important question in this domain is how learning affects the overall problem-solving task, namely how it improves the speed and reliability of the design optimization process. Does learning make the design process faster or slower? Are the resulting designs better or worse? To measure these effects, we performed optimizations for 25 new randomly generated goals using the formulations suggested by each learning method. Table 12 shows the effect that C4.5 (with pruning) and MFC had on the average course time (the quality of the design), and average number of evaluations (the speed of the optimization), as compared with the "old way" of doing optimization without incorporating any of the three constraints into the operators. The first column in the table shows the percentage difference between the optimized course-time produced with the original formulation, and the optimized course time produced with the specified formulation. The second column shows the percentage difference between the cost of performing the optimization with the original formulation, and the cost of performing it with the specified formulation.

TABLE 12. *Effect of using formulations chosen by learner on optimization performance. A positive quality change indicates an improvement in quality (which is a reduction in course time).*

method	quality change	CPU time change
omniscient	+0.085%	-36%
exhaustive	+0.085%	+384%
C4.5	+0.080%	-35%
MFC	+0.029%	-32%
none	0	0
random	-0.276%	-40%
all	-0.599%	-74%

We also include in this table the performance of several other methods. A hypothetical "omniscient" problem solver always magically guesses the best possible choice (the one that results in the best course time).[15] No learning method will enable results superior to those produced by this method. The "exhaustive" optimization method performs eight optimizations for each goal, using all eight possible formulations, and then chooses the best resulting design. Incorporating "all" constraints all the time results in the fastest possible optimization within this set of formulations (at the cost of quality loss).

C4.5 produced a significant speedup in optimization, with no quality loss. In fact, it produced a small quality increase. (This quality increase suggests that with the original formulation, the optimizer gets "stuck" on the "ridges" that the constraints cause the search space to have, and therefore sometimes fails to get the optimum.) MFC produced a slightly smaller speedup and a slightly smaller quality improvement. The difference between C4.5 and MFC in quality change was, however, statistically significant at the 99% confidence level, according to the paired t-test. Both learning methods performed substantially better than random guessing. C4.5 performed almost as well as the hypothetical omniscient learner, which means it performed almost as well as any learner could possibly do.[16]

Incorporating all of the constraints all of the time resulted in a very large speedup, with a modest quality loss. This method may be appropriate if one wants a quick and approximate optimization. It might, for example, be used in the early stages of design when the engineer wants to get a feel for the search space by asking "what-if" questions.

One question that these results raise is how training-data quantity affects performance. If one does not have results from a large number of previous optimizations available, then one can either run some extra optimizations to generate training data (which is expensive), or do the learning with less training data (which is likely to produce higher error rates and lower optimization performance). We ran some experiments to determine how C4.5's performance varies

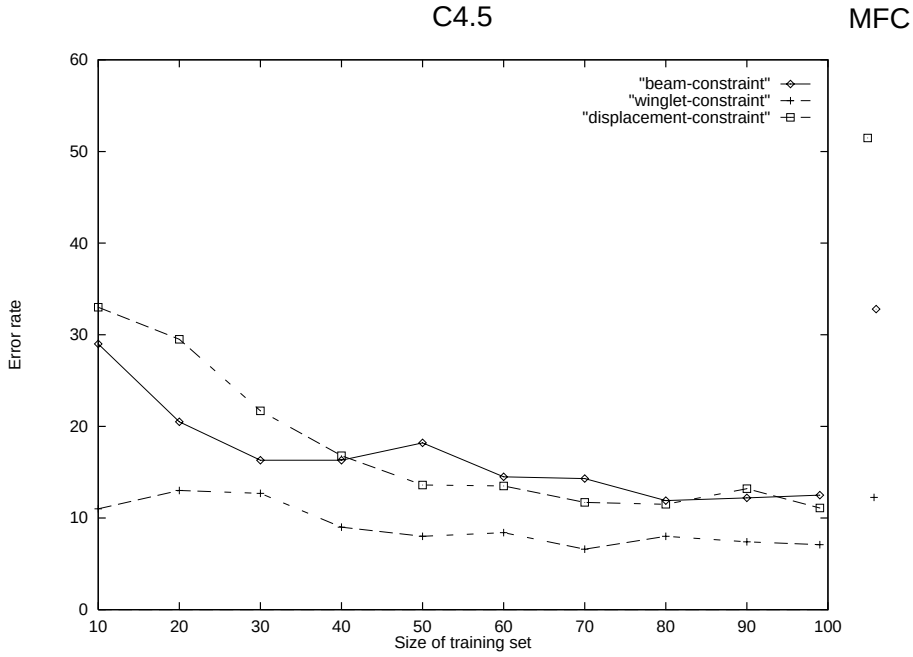

Figure 9. Effect of training set size on learner performance.

with training-set size, and how its performance compares with that of MFC for various training-set sizes. We applied our learning approach to datasets of varying sizes, with the error rates shown in Figure 9. For each training-set size in the figure, we randomly chose 10 different subsets of our training data of that size, and performed 10-fold cross-validation on each subset. The figure shows the averages. The three symbols at the right side of the figure show MFC's performance on the full training set. C4.5 outperformed MFC for every training-set size, but C4.5's error rate on smaller training sets was significantly larger than C4.5's error rate for larger training sets (with performance reaching an asymptote for training sets of about 60 cases or more).

Formulation Selection Results in Airframe Domain

We believe that our formulation selection technique is applicable to a broad range of design optimization problems. To test the domain-independence of the formulation selection technique, we performed additional experiments in the airframe domain, and compared the impact on optimization performance of C4.5 with that of MFC.

In the airframe domain, there are eight design parameters, each of which can have an upper and lower bound. The optimal design sometimes lies at the bounds of some of these parameters, depending on the mission.

We used CFSQP as the optimizer, and used the same simulator and the

```
overland <= 95.0872% : zero (54.0)
overland > 95.0872% :
|   takeoff = no: zero (12.0/1.0)
|   takeoff = yes: nonzero (10.0)
```

Figure 10. Learned decision tree for predicting if the taper ratio will be at its lower bound of zero.

TABLE 13. *Cross-validated error rates for selecting whether to incorporate each lower bound, in the airframe domain.*

design parameter	C4.5	MFC	Random
wing taper ratio	2.7%	14.5%	50.0%
wing sweep	2.5%	27.6%	50.0%
fuselage taper length	3.9%	22.4%	50.0%
fuel annulus width	13.6%	5.3%	50.0%

same space of missions as in the prototype synthesis experiments. We used the same C4.5 decision tree to predict which missions are feasible. As training data, we used the same 100 10-point random multistart CFSQP optimizations, 76 of which are feasible.

We used the 76 feasible missions to train C4.5 for formulation selection. Of the eight design parameters, four were never at their upper or lower bounds at the apparent optima for any of the 76 missions. The other four had optima at their lower bounds for some missions. We trained C4.5 to predict whether these four design parameters would be at their lower bounds, depending on the mission. C4.5 produced a separate decision tree for each of these four design variables. For example, Figure 10 shows the decision tree for wing taper ratio. This decision tree says that wing taper ratio will be at its lower bound of zero, unless the mission includes a takeoff phase and is almost entirely over land. The four decision trees can be used to select among 16 (2^4) possible formulations.

Table 13 compares the cross-validated error rates of C4.5 with those of most frequent class and random guessing for each of the four design parameters. For the first three parameters, C4.5 did much better than most frequent class. For the fourth parameter, fuel annulus width, C4.5 did much worse than most frequent class, violating our expectations. In this case, only 4 of the 76 training examples were positive examples. We suspect C4.5 would need more training examples to be more accurate. Interestingly, in our prototype synthesis experiments, CART had difficulty predicting the optimal value of fuel annulus width.

To determine the impact of the formulations selected by the various methods on optimization performance, we randomly generated 25 new missions. Table 14 compares the performance of the various methods of formulation selection when doing optimizations for these new missions. For the methods

TABLE 14. *Effect of using formulations chosen by learner on optimization performance, in airframe domain.*

method	success	time change
omniscient	16	-51%
exhaustive	16	+1206%
C4.5/none	15	-36%
none	15	0
C4.5/MFC	13	-57%
MFC	13	-21%
all	3	-55%

that used C4.5, we used the decision tree of Figure 6 to predict whether each new mission was feasible, and only performed optimizations for those missions that were predicted to be feasible. For the other methods, we performed optimizations for all 25 missions. Each optimization was a 10-point multistart. The "success" column indicates for how many of the missions the specified method came within 1% in takeoff mass of the best design found.[17] The "time change" column shows the change in total number of simulations used in all of the optimizations performed, compared with not incorporating any constraints.

Because cross-validation showed that C4.5 under-performs MFC for predicting whether to incorporate fuel annulus width, we did not use C4.5 to decide whether to incorporate this parameter. We used C4.5 to decide whether to incorporate the other three parameters, and used two different methods to decide whether to incorporate fuel annulus width. The first method used MFC to decide whether to incorporate the fuel annulus width, which resulted in always incorporating it. The results of this method are labeled "C4.5/MFC" in Table 14. For the second method, we decided to play it safe and never incorporate fuel annulus width, since cross validation suggests that we are not able to accurately predict when this parameter will be at its bound. The results of this method are labeled "C4.5/none" in Table 14. We compare these methods with most frequent class, and with the exhaustive method that does optimizations for all 16 (2^4) formulations, and the omniscient method which magically guesses the best formulation.

The first interesting thing to note about Table 14 is that there is one mission for which CFSQP failed to reach the optimum without reformulation. The only way to reach the optimum for this mission is to use the "omniscient" method (which does not exist), or the "exhaustive" method (which is extremely expensive). The next thing to note is that using the formulations selected by C4.5 for the first three parameters, while not incorporating fuel annulus width ("C4.5/none"), reduces cost by 36% compared with not incorporating any constraints ("none"), without any loss of quality. Using C4.5 for the first three

parameters, and MFC for fuel annulus width (C4.5/MFC), causes CFSQP to fail to find the optimum in two additional cases. Using MFC for all parameters causes the same number of missed optima, at a higher cost. And incorporating all of the parameter bounds all of the time results in CFSQP almost always failing to get to the optimum.

The airframe domain results are surprisingly similar to the yacht domain results. In the yacht domain, using the formulations selected by C4.5 reduced the cost of optimization by 35% (Table 12), while in the airframe domain the speedup was 36%. In the yacht domain, using C4.5 also resulted in a small quality increase, while in the airframe domain, quality remained the same. This may be because the yacht domain reformulations increase the smoothness of the search space (by eliminating the 12m-rule penalties), while the airframe domain reformulations do not. Another interesting thing to note is that while the difference between MFC and C4.5 was small (but statistically significant) in the yacht domain, it was much larger in the airframe domain.

RELATED WORK

Cerbone (Cerbone, 1992) has reported work which applied machine-learning techniques to a problem similar to our prototype-selection problem. His design space, in the domain of truss design, has an exponential number of disconnected search spaces. He uses inductive learning techniques to learn rules for selecting a subset of these search spaces for further exploration. In contrast, our system has a smaller number of prototypes (each of which defines a search space) from which to choose, and it just chooses one of them. Cerbone uses an ad-hoc utility function to combine solution quality and search time when evaluating his learning methods, while we only consider solution quality in this chapter. Cerbone also presents two learners that incorporate background knowledge by incorporating the objective function into the learner.

Research on prototype-retrieval strategies for hill-climbing design optimization is reported by Ramachandran *et al.* (Ramachandran et al., 1992), who investigated a number of library-based methods for finding starting points for the DPMED iterative parameter-design system. These included a nearest-neighbor method, a curve-fitting method, and a hybrid method. The curve-fitting method is similar to our prototype synthesis method. It uses regression to find a function mapping goal parameters to initial design parameters, whereas our approach uses inductive learning to find a regression tree mapping goal parameters to initial design parameters. Ramachandran *et al.* compared their retrieval strategies in terms of the numbers of iterations needed to carry out the hill-climbing design-optimization process. They showed that starting points obtained by curve fitting led to fewer iterations than were required when the nearest-neighbor method was used. In contrast to this work, our work has evaluated retrieval strategies in terms of the quality of the resulting designs, in addition to the number of iterations needed to find them.

Researchers in case-based reasoning have investigated the use of library-

retrieval techniques for case-based design (Sycara and Navinchandra, 1992; Kolodner, 1993), but have not used them to initialize an iterative design process. (Bhatta and Goel, 1995) describe a system that learns to retrieve a starting point for the design of a high-acidity sulfuric acid cooler. They evaluate the performance of this indexing system based on its effect on retrieval time, and not based on its impact on optimization performance.

In (Rasheed and Hirsh, 1999), Rasheed and Hirsh describe a *screening module* that they added to a genetic algorithm used for engineering design optimization. The screening module uses a nearest-neighbor approach to learn to avoid evaluating points in the search space that are likely to have poor evaluation functions because they are near other "bad" points. They tested their algorithm in multiple domains, including the supersonic aircraft domain described in this chapter, and found that the screening module makes the optimization significantly faster.

Gelsey *et al.* (Gelsey et al., 1996) describe a Search Space Toolkit which assists in determining properties of the search space that can be used for reformulation. (Choy and Agogino, 1986) describe a system that automates (Papalambros and Wilde, 1988)'s method of using monotonicity analysis to detect constraint activity.

In (Williams and Cagan, 1994), Williams and Cagan present *activity analysis*, a technique inspired by monotonicity analysis. Their technique is similar to the formulation selection technique described in this chapter, except that they use qualitative reasoning instead of machine learning to determine which constraints will be active at the optimum. Their technique has the advantages that it does not require training data, and that the reformulation is guaranteed not to lose the global optimum. It has the disadvantage that it requires that the objective function and constraint functions be symbolically differentiable and composed of simple arithmetic operations; it would therefore not be applicable to the complex simulators used in the experiments described in this chapter.

A number of research efforts have combined AI techniques with numerical optimization. (Ellman et al., 1993) describes a method for switching between a less expensive, less accurate simulator, and a more expensive, more accurate simulator during optimization, based on the magnitude of the gradient. (Bouchard et al., 1988) describes ways in which expert systems could be applied to the parametric design of aeronautical systems. (Hoeltzel and Chieng, 1987) describe a system for digital chip design in which design is done at an abstract level, using machine learning to estimate the performance that would be obtained if the design were carried out at a more detailed level. (Orelup et al., 1988) describes a system called Dominic II that uses an expert system to switch among various strategies during numerical optimization. None of these efforts is focused directly on the problems of prototype selection and formulation selection addressed in this chapter.

Simulated annealing (SA) and genetic algorithms (GA) are able to deal with certain pathologies, such as nonsmoothness, but they tend to be much slower than gradient-based optimization. They tend to require thousands, or even tens

of thousands, of simulations, and are thus not practical when each simulation is expensive.

Powell (Powell, 1990; Tong et al., 1992; Powell and Skolnick, 1993) has built a module called Inter-GEN, part of the ENGINEOUS system (Tong, 1988), that seeks to combine the ability of genetic algorithms to handle multiple local optima with the speed of numerical optimization algorithms. It contains a genetic algorithm, and a numerical optimizer, and uses a rule-based expert system to decide when to switch between the two. Powell has tested his system on a realistic jet engine design problem. He does not, however, address the issues of prototype selection or formulation selection.

FUTURE WORK

This chapter presents an initial exploration of the use of inductive learning to set up an optimization, and there are a number of directions for future work. These directions for future work fall into three groups: extending this work to more difficult design tasks, improving results by using other learning methods, and applying inductive learning to other choices that must be made in setting up an optimization.

Other Design Tasks

First, the experiments reported here explore the sensitivity of our prototype-selection method to the nature of the design library, specifically with respect to the quality of the stored designs. It would be helpful to more fully explore the sensitivity of our approach to the design library, for example by studying how our approach scales up as the library size increases.

The yacht domain results presented here apply to a constrained class of yacht-design goals, those comprised of a single leg (for formulation selection) or two legs (for prototype selection). One question is how this approach can be applied to courses comprised of varying numbers of legs. We believe that we could get reasonable optimization performance by using the trees learned from single-leg courses to perform multi-leg formulation selection in the following way: If a constraint should be incorporated for every leg of the race-course, then incorporate it for the full, multi-leg course. We need to test how well optimization performs when handling race-courses in this manner. We could also attempt to learn directly for multi-leg race-courses. Doing so would raise an interesting machine-learning question, since describing a multi-leg race-course requires a variable number of attributes, and thus traditional learners such as C4.5 do not directly apply. Learning methods operating on more expressive representations, such as inductive logic programming systems like FOIL (Quinlan, 1990), may enable going beyond the simple representation of goals used here and handling more complicated goals, including those involving multi-leg race courses or multiple disciplines.

In the results presented here, we assume that the only change between the previous design sessions and the current design session is the design goal (for example, expressed as a $(\textit{wind speed}, \textit{heading})$ pair for formulation selection in the yacht domain). An interesting question is what would happen if in addition to changing the goal, we also changed the constraints, or the simulator, or the form of the goal. We would need to find a way to encode as a set of attributes for the learner whatever had changed.

We believe that the formulation selection results presented here will easily generalize to situations in which there are more than sixteen formulations. We used the results from the same set of 100 optimizations to perform three separate learning tasks (for three constraints), and then combined the rules generated by these three learning sessions to select one of the eight formulations. As the number of formulations grows, the number of constraints, and therefore the amount of CPU time needed for the learning, will grow logarithmically with the number of formulations. The CPU time needed for learning is currently insignificant compared with the CPU time needed for the subsequent optimizations. We expect that as the number of formulations grows, the number of training examples needed will remain constant (since the same training examples are used for each constraint), and the amount of CPU time needed for learning will remain insignificant. We plan to test this hypothesis by using other constraints within the yacht design domain, such as the "boat doesn't sink" constraint.

The learning approach could also be used to decide when to reformulate soft constraints as hard constraints. If it were known with a high degree of confidence that a certain soft constraint will not be violated at the optimum for certain goals, then this soft constraint could be converted into a hard constraint for those goals, which would eliminate a ridge from the search space and thereby make optimization more robust (although it would not reduce the dimensionality of the search space). For example, in the training data that we collected, the *beam constraint* was never violated, so it might be replaced safely with a hard constraint.

Other more-difficult problems might involve a less-smooth search space, a higher-dimensional goal space, or a less reliable optimizer. Such problems may arise when we test this method in still other domains.

Other Learning Methods

We found that C4.5 performed nearly as well as a hypothetical "omniscient" learner, when doing formulation selection for the fairly simple design problems that we used in our experiments. When doing prototype selection, however, there was room for improvement. Other learning methods might prove useful in attacking the prototype selection problem, and might also prove useful when doing formulation selection for some of the harder design problems described in the previous subsection. For example, it would be interesting to see how well neural networks, nearest-neighbor methods, or statistical regres-

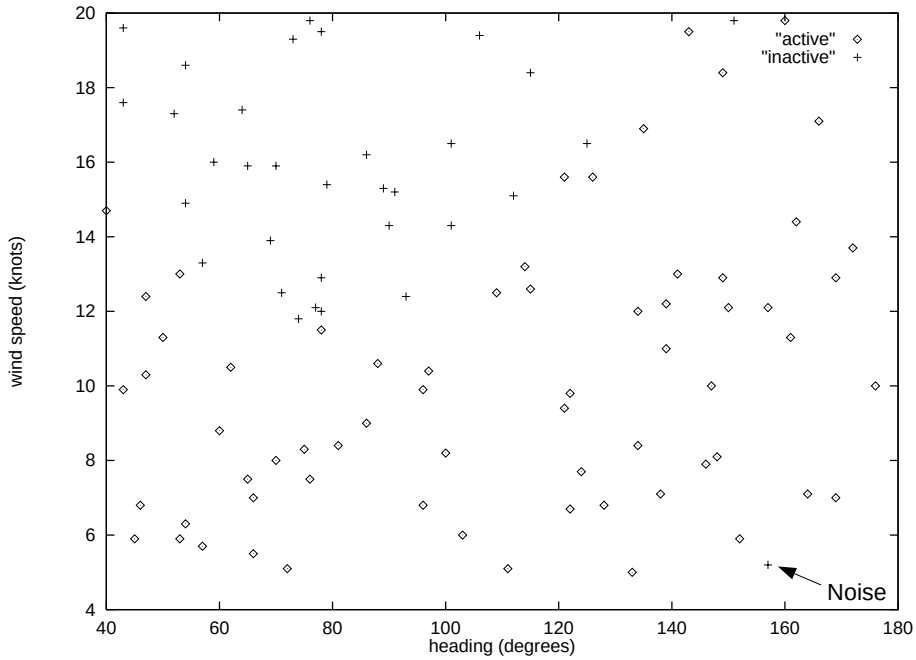

Figure 11. Activity of the beam constraint over the goal space.

sion would perform. In particular, C4.5, like most decision-tree learners, uses linear, axis-parallel cuts in its decision trees. However, Figure 11 shows how the activity of the beam constraint varies over the goal space in the training data we used — the space is clearly divided into two regions (except for one point which we believe is noise). The border between these regions does not appear to be axis parallel, and appears to be nonlinear. This suggests that better formulation-selection performance might be achieved using an "oblique" decision tree learner, such as OC1 (Murthy et al., 1994), or by attempting to learn nonlinear region boundaries.

As would be expected, even though our yacht-domain formulation-selection results with C4.5 were nearly optimal for 100 examples, results degrade when given less training data. Although it would be interesting to see if other learning methods would have better small-dataset performance, for any learner we would expect performance to be inferior for small enough datasets. One approach for improving results in such small-dataset cases — as well as in other cases where off-the-shelf learners such as C4.5 may not perform well even if given larger datasets — is to integrate background knowledge into the learning process. One form of background knowledge that is often available, such as in the yacht-design domain, is *modality constraints*. This is knowledge that expresses the modality of the learned class with respect to the attributes. For example, we believe that optimal *beam* is monotonically increasing in wind speed, and monotonically decreasing in heading. We also know that the activity of any constraint of the form $f(x_1, x_2, \ldots, x_n) \leq k$ must be monotonic

in k, so, for example, the activity of a cost constraint must be monotonic in the cost threshold. One open question is how such knowledge could be integrated into learning. One approach would be to use such modality constraints to remove from the training data points that violate the constraints (on the assumption that these points are noise). A second approach is to modify the tree induction algorithm so that it will never construct a tree that violates the constraints. A similar approach was used to constrain decision lists in (Clark and Matwin, 1993).

Finally, even after our learning approach is applied, every additional future optimization can serve as an additional training point for the learning. Thus learning methods that can work in an incremental fashion might also prove useful for this task. In addition, it may prove useful to develop methods that select suitable data prior to learning. For example, when there are not enough existing optimizations to achieve adequate learning results, additional optimizations can be performed to generate further training data. Rather than performing these new optimizations for random goals or for a set of goals that span the goal space, one could allow the learner to choose the goals to be used in the new training data. Background knowledge — such as modality constraints — could prove particularly useful in selecting such goals.

Other Setup Choices

We have applied inductive learning to several decisions that must be made when setting up an optimization, including choosing a starting protype and a formulation of the search space, and predicting whether a design goal is achievable. There are other parts of the setup process to which inductive learning might be applicable. For example, one might try to use inductive learning to choose an optimization algorithm, or a good value of the optimizer's stopping tolerance, or a good step size to use in gradient computation, or a good box within which to randomly generate starting prototypes, or a good number of random starting prototypes to generate, or the right level of accuracy to use in the simulator. For each of these decisions, it would need to be determined whether the best choice depends on the design goal. Finally, more experiments need to be done to explore the impact on optimization performance of using inductive learning to simulataneously make multiple choices within the optimization setup problem.

CONCLUSION

Gradient-based methods do not perform well when optimizing designs using simulators that have pathologies. We have described and demonstrated the utility of four techniques that improve optimization performance in such situations by using inductive learning to make decisions when setting up the design optimization. Two of these are methods of choosing an initial prototype for optimization. Prototype selection is especially appropriate in domains such as the

yacht domain in which there is a database of previous designs available, and the available simulators are noisy. Prototype synthesis is especially appropriate in domains such as the aircraft domain in which finding a feasible design is difficult. The third technique, feasible goal prediction, is similarly useful in such a domain.

We tested the fouth technique, formulation selection, in both the yacht domain and the aircraft domain. We showed that using this technique can make design optimization faster, because the reformulation reduces the dimensionality of the search space, and more reliable, because the reformulation can make the search space smoother.

NOTES

[1] The cost of generating this table is discussed in the "Cost of Learning" subsection.

[2] CART stands for Classification And Regression Trees

[3] We call the simulator *multidisciplinary* because it contains code to evaluate the design using several engineering disciplines. For example, our aircraft simulator includes weights, aerodynamics, and propulsion.

[4] In 1992, the 12-meter class was replaced with the new America's Cup Class.

[5] This is the boat that won the 1987 America's Cup competition, returning the trophy to the United States after an Australian win in 1983 (Letcher et al., 1987).)

[6] The four operators we chose were *Scale-X*, *Scale-Y*, *Prism-Y*, and *Scale-Keel*. We chose these operators because the results of our earlier work on operator-importance analysis suggested that these are the four most important operators (Ellman and Schwabacher, 1993).

[7] CFSQP stands for "C code for Feasible Sequential Quadratic Programming."

[8] A quadratic programming problem consists of a quadratic objective function to be optimized, and a set of linear constraints.

[9] Some randomly generated designs, which we call "unevaluable points," cannot be simulated, either because the designs are meaningless or because of limitations of the simulator.

[10] Because CFSQP failed to find a feasible point in some of these optimizations, it was not possible to compute the average design quality.

[11] The simulator assumes that there is perfect reefing, so additional sail area can never hurt the yacht's performance.

[12] Because we incorporated the 12-Meter rule into the simulator, we did not need to use it as an explicit constraint.

[13] Although this figure shows only a "snapshot" of the search space for specific values of the other design parameters, we believe that the trend shown in the figure is generally applicable.

[14] Operators containing numerical solvers would probably be more computationally expensive than operators containing the algebraic solutions of the constraint functions, so the CPU time savings from reformulation would probably be smaller.

[15] We simulated the omniscient learner by performing optimizations using all eight formulations for each goal (as in the "exhaustive" method), and then ignoring the cost of the seven optimizations that turned out not to be best.

[16] Interestingly, according to the t-test, the difference between C4.5 and the omniscient method was not statistically significant, but this just illustrates a limitation of the t-test, since we know that the omniscient method really is better, on average, than C4.5.

[17] Because CFSQP failed to find a feasible point in some of these optimizations, it was not possible to compute the average design quality.

ACKNOWLEDGMENTS

This research has benefited from numerous discussions with members of the Rutgers HPCD project. We especially thank Gerard Richter for his contributions to the formulation selection work, Andrew Gelsey for helping with the cross-validation code, John Keane for helping with RUVPP, and Andrew Gelsey, Brian Davison, and Tim Weinrich for comments on previous drafts of this chapter. This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA-funded NASA grant NAG 2-645 and under contract ARPA-DABT 63-93-C-0064. Mark Schwabacher was a Postdoctoral Research Associate at the National Institute of Standards and Technology for a portion of the time that he worked on this research; during that time he was supported by a National Research Council Postdoctoral Research Associateship. This chapter was previously published in a slightly different version as a journal article in *AI EDAM* 12:2; we thank Cambridge University Press for giving us permission to republish it here.

CURRENT ADDRESSES

Mark Schwabacher
NASA Ames Research Center
MS 269-1
Moffett Field, CA 94035

Thomas Ellman
Department of Computer Science
Vassar College
Poughkeepsie, New York 12601

REFERENCES

Bhatta, S. and Goel, A., Model-based design indexing and index learning in engineering design, in *Working Notes of the IJCAI Workshop on Machine Learning in Engineering*, 1995.

Bouchard, E. E., Kidwell, G. H., and Rogan, J. E., The Application of Artificial Intelligence Technology to Aeronautical System Design, in *AIAA/AHS/ASEE Aircraft Design Systems and Operations Meeting*, Atlanta, Georgia, 1988, AIAA-88-4426.

Breiman, L., Classification And Regression Trees, Belmont, Calif.: Wadsworth International Group, 1984.

Cerbone, G., Machine learning in engineering: Techniques to speed up numerical optimization, Technical Report 92-30-09, Oregon State University Department of Computer Science, 1992, Ph.D. Thesis.

Char, B., Geddes, K., Gonnet, G., Leong, B., Monagan, M., and Watt, S., First Leaves: A Tutorial Introduction to Maple V, Springer-Verlag and Waterloo Maple Publishing, 1992.

Choy, J. and Agogino, A., SYMON: Automated Symbolic Monotonicity Analysis System for Qualitative Design Optimization, in *Proceedings ASME International Computers in Engineering*

Conference, 1986.

Clark, P. and Matwin, S., Using qualitative models to guide inductive learning, in *Proceedings of the tenth international machine learning conference*, 49–56, Morgan Kaufmann, 1993.

Ellman, T., Keane, J., and Schwabacher, M., The Rutgers CAP Project Design Associate, Technical Report CAP-TR-7, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1992, ftp://ftp.cs.rutgers.edu/pub/technical-reports/cap-tr-7.ps.Z.

Ellman, T., Keane, J., and Schwabacher, M., Intelligent Model Selection for Hillclimbing Search in Computer-Aided Design, in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 594–599, Washington, DC: MIT Press, Cambridge, MA, 1993.

Ellman, T. and Schwabacher, M., Abstraction and Decomposition in Hillclimbing Design Optimization, Technical Report CAP-TR-14, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1993, ftp://ftp.cs.rutgers.edu/pub/technical-reports/cap-tr-14.ps.Z.

Gelsey, A., Schwabacher, M., and Smith, D., Using Modeling Knowledge to Guide Design Space Search, AI Journal, 101(1-2), 35–62, 1998.

Gelsey, A., Smith, D., Schwabacher, M., Rasheed, K., and Miyake, K., A Search Space Toolkit: SST, Decision Support Systems, 18, 341–356, 1996.

Hoeltzel, D. and Chieng, W., Statistical Machine Learning for the Cognitive Selection of Nonlinear Programming Algorithms in Engineering Design Optimization, in *Advances in Design Automation*, Boston, MA, 1987.

IYRU, The Rating Rule and Measurement Instructions of the International Twelve Metre Class, International Yacht Racing Union, 1985.

Kolodner, J., Case-Based Reasoning, San Mateo, CA: Morgan Kaufmann Publishers, 1993.

Lawrence, C., Zhou, J., and Tits, A., User's Guide for CFSQP Version 2.3: A C Code for Solving (Large Scale) Constrained Nonlinear (Minimax) Optimization Problems, Generating Iterates Satisfying All Inequality Constraints, Technical Report TR-94-16r1, Institute for Systems Research, University of Maryland, College Park, MD, 1995.

Letcher, J., The Aero/Hydro VPP Manual, Southwest Harbor, ME: Aero/Hydro, Inc., 1991.

Letcher, J., Marshall, J., Oliver, J., and Salvesen, N., Stars and Stripes, Scientific American, 257(2), 1987.

Murthy, S., Kasif, S., Salzberg, S., and Beigel, R., A System for Induction of Oblique Decision Trees, Journal of Artificial Intelligence Research, 2, 1–32, 1994.

Orelup, M. F., Dixon, J. R., Cohen, P. R., and Simmons, M. K., Dominic II: Meta-Level Control in Iterative Redesign, in *Proceedings of the National Conference on Artificial Intelligence*, 25–30, St. Paul, MN: MIT Press, Cambridge, MA, 1988.

Papalambros, P. and Wilde, J., Principles of Optimal Design, New York, NY: Cambridge University Press, 1988.

Powell, D., Inter-GEN: A Hybrid Approach to Engineering Design Optimization, Ph.D. thesis, Rensselaer Polytechnic Institute Department of Computer Science, Troy, NY, 1990.

Powell, D. and Skolnick, M., Using genetic algorithms in engineering design optimization with non-linear constraints, in *Proceedings of the Fifth International Conference on Genetic Algorithms*, 424–431, Univeristy of Illinois at Urbana-Champaign: Morgan Kaufmann, Los Altos, CA, 1993.

Quinlan, J. R., Learning logical definitions from relations, Machine Learning, 5, 239–266, 1990.

Quinlan, J. R., C4.5: Programs for Machine Learning, San Mateo, CA: Morgan Kaufmann, 1993.

Ramachandran, N., Langrana, N., Steinberg, L., and Jamalabad, V., Initial Design Strategies for Iterative Design, Research in Engineering Design, 4, 159–169, 1992.

Rasheed, K. and Hirsh, H., Learning to be Selective in Genetic-Algorithm-Based Design Optimization, Artificial Intelligence for Engineering Design, Analysis, and Manufacturing, 13, 1999.

Rogers, D. and Adams, J., Mathematical elements for computer graphics, McGraw-Hill, second edition, 1990.

Schwabacher, M., The Use of Artificial Intelligence to Improve the Numerical Optimization of Complex Engineering Designs, Technical Report HPCD-TR-45, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1996, Ph.D. Thesis. http://www.cs.rutgers.edu/∼schwabac/thesis.html.

Schwabacher, M. and Gelsey, A., Intelligent Gradient-Based Search of Incompletely Defined Design Spaces, Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 11(3), 199–210, 1997.

Sycara, K. and Navinchandra, D., Retrieval Strategies in a Case-Based Design System, in C. Tong and D. Sriram, editors, *Artificial Intelligence in Engineering Design (Volume II)*, 145 – 164, New York, NY: Academic Press, 1992.

Tong, S. S., Coupling Symbolic Manipulation and Numerical Simulation for Complex Engineering Designs, in *International Association of Mathematics and Computers in Simulation Conference on Expert Systems for Numerical Computing*, Purdue University, West Lafayette, IN, 1988.

Tong, S. S., Powell, D., and Goel, S., Integration of Artificial Intelligence and Numerical Optimization Techniques for the Design of Complex Aerospace Systems, in *1992 Aerospace Design Conference*, Irvine, CA, 1992, AIAA-92-1189.

Williams, B. and Cagan, J., Activity Analysis: The Qualitative Analysis of Stationary Points for Optimal Reasoning, in *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1217–1223, Seattle, WA: MIT Press, 1994.